(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 April 2001 (12.04.2001)

PCT

(10) International Publication Number
WO 01/26336 A2

(51) International Patent Classification[7]: H04L 29/06

(21) International Application Number: PCT/US00/27756

(22) International Filing Date: 6 October 2000 (06.10.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/158,409 7 October 1999 (07.10.1999) US

(71) Applicant *(for all designated States except US)*: XBIND, INC. [US/US]; Suite 13C, 55 Broad Street, New York, NY 10004 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: ADAM, Constantin [RO/US]; Apartment 4D, 9801 Shore Road, Brookly, NY 11209-7633 (US). LAZAR, Aurel, A. [DE/US]; Apartment 32A, 410 Riverside Drive, New York, NY 10025 (US). LIM, Koon-Seng [SG/US]; Apartment 53A, 419 West 118th Street, New York, NY 10027 (US).

(74) Agent: NUGENT, Elizabeth, E.; Choate, Hall & Stewart, 53 State Street, Boston, MA 02109 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:

— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: CONFIGURATION INFRASTRUCTURE IN SUPPORT OF BOOTING AND SEAMLESS ATTACHMENT OF COMPUTING DEVICES TO MULTIMEDIA NETWORKS

(57) Abstract: A configuration infrastructure that allows computing devices to dynamically bootup and attach to a multimedia network domain. The configuration infrastructure includes a centralized domain configuration tree for organizing and sharing the default domain configuration, management and control information in a uniform and consistent manner; a domain directory service that publishes and retrieves the configuration information and registers any devices attached to the domain; a set of local configuration trees (one per device) that allow to cache the device configuration settings and remember the customized configuration attributes; a general design template for building software controllers so that they can be integrated with our configuration infrastructure; a general protocol for enabling controller-to-controller negotiation and delegation; a software distribution and downloading service for remote installation of controllers and system components; a watchdog controller that continually monitors the location and state of attachment of the client; and a control and monitoring system for the bootup and attachment processes.

WO 01/26336 A2

Configuration Infrastructure in Support of Booting and Seamless Attachment of Computing Devices to Multimedia Networks

This application claims benefit of priority of U.S. Provisional Application No. 60/158,409, filed October 7, 1999, which is incorporated herein by reference.

Field of the Invention

The present invention relates to systems that provide capabilities for automatic booting or dynamic attachment of devices or software objects.

Background of the Invention

Computer programs may require a set of initialization parameters at execution time. These parameters can be provided locally, by typing them on a command line, or using a script. This approach works well for a single program, or for a small set of applications running on one or several computers. However, it does not scale well for a network domain, where a larger number of instances of the same service need to be started with the same initial configuration parameters.

U.S. Patent No. 5,852,722 describes a system and method for automatic configuration of home network computers. The network computers determine whether they have the necessary information to boot up and, if they do not, they request this information from a configuration server. The provided method is useful only for initial booting up of computers from a single service provider. U.S. Patent No. 5,859,978 describes a method for application programs management. An application launcher looks up a configuration directory, sets up the resources needed by applications, and automatically launches them.

Neither of these patents addresses the issue of distributed applications. Their main objective is to configure a local entity (*e.g.*, a computer or an application), but they do not address at the configuration stage the way in which this entity will interact with its peers in the network.

The Jini architecture is based on lookup and discovery services. In this framework, a device auto-configures itself, and registers with a lookup service. When an entity in the network needs to use the service provided by another entity, it locates

it by querying the lookup service. Then it will contact the other entity, and typically will download the code that will allow it to run the desired service.

The present invention uses directories to store configuration information. Two known models of directory access that are suitable for use with the invention are therefore briefly described.

In the X.500 model, a directory is a function that maps a name to a set of attributes of an object. Directories provide distributed services whose components are called Directory System Agents (DSA). A Directory Service Protocol (DSP) is used for communication between DSAs. The data is stored in a Directory Information Tree (DIT). The nodes of the tree are attributes of an object belonging to a specific class. Every attribute is represented as an Attribute Value Assertion (AVA), containing an attribute type and an attribute value. The Knowledge Information Tree (KIT) mirrors the DITs for which each DSA is responsible and the relationship between DSAs.

For a centralized system, a name is mapped to an object in a table. For a distributed system, a name is searched for in two steps. First, the server that contains the name mapping is located. Second, the attributes are retrieved from the server.

Finally, a Directory User Agent (DUA) is a service interface providing directory access to users. The Directory Access Protocol (DAP) represents the means for a DUA to request services from a DSA.

The Lightweight Directory Access Protocol (LDAP) was proposed in order to provide a much simpler way to access a directory than X.500. LDAP is carried directly over TCP, thereby eliminating the X.500 session/presentation layers overhead. In addition, most names are encoded as ordinary strings, as opposed to the Attribute Value Assertions of X.500. This further simplifies the protocol, as one does not need to take into account the type of each attribute.

LDAP is a directory access protocol that provides both read and update access. The Internet Engineering Task Force (IETF) is currently standardizing the version 3 of the LDAP protocol. Clients transmit protocol requests describing the operation to be performed to a server. The server performs the necessary operations and then returns a response containing any results or error codes to the client. The client structure is as simple as possible. The security protocol to be used with LDAP is still

under development. The protocol also contains a referral mechanism that might redirect clients to other servers.

In LDAP, one or several servers provide access to a directory information tree, made up of entries. The entries have names, and consist of a set of attributes. One or several attribute names from the tree form a Relative Distinguished Name (RDN), which must be unique among all the entry siblings.

A schema that enforces the object class definitions and attribute syntax constraints is associated with each LDAP directory. The schema allows an LDAP server to check type correctness in add or update operations, as well as to interpret a compare operation.

The Directory Enabled Network is a noteworthy industry initiative launched by Cisco and Microsoft. A directory enabled network is a network where user profiles, applications and network services are integrated through a common information model that stores network state and exposes network information. This information then enables bandwidth utilization to be optimized; it enables policy-based management and it provides a single point of administration of all network resources. The Directory Enabled Network relies on LDAP to exchange data between the directory and the clients. In addition to LDAP, Microsoft has introduced the Active Directory Service Interface (ADSI), that allows a client to access the directory using several protocols, LDAP being a possible protocol instantiation. The Directory Enabled Network specifies also a schema, into which any managed object description should fit.

Summary of the Invention

The present invention provides a method of booting up or attaching a computing device to a multimedia network by providing a configuration infrastructure for a set of software entities called controllers. The configuration infrastructure may comprise a centralized domain configuration tree for organizing and sharing the default domain configuration, management and control information in a uniform and consistent manner; a domain directory service that publishes and retrieves the configuration information; a set of local configuration trees (one per device) that allow to cache the device configuration settings and remember the customized

configuration attributes; a general design template for building software controllers so that they can be integrated with our configuration infrastructure; a general protocol for enabling controller-to-controller negotiation and delegation; a software distribution and downloading service for remote installation of controllers and system components; a watchdog controller that continually monitors the location and state of attachment of the client; and a control and monitoring system for the bootup and attachment processes.

The configuration infrastructure may provide the capabilities including automatically booting up or integrating an already running controller into a multimedia network; publishing a new service in a multimedia network, which may be a centralized or a locally distributed service; and monitoring the bootup and/or integration processes. Further, it may provide a modular configuration management structure for different service types, a controller design template to ensure controllers will seamlessly integrate into and interact with the rest of the configuration system, and a negotiation and delegation protocol for controllers.

A configuration information tree may be structured in a way that allows it to cache the last configuration of a controller instance, thus saving the preferences set during a session; to store the membership of a group of controllers that are launched together and the configuration information associated with the controller group; to provide a template configuration for any controller or group of controllers that can be launched in the network, and/or to store the configuration of newly advertised services.

A domain directory service may be also provided that represents a front end interface to the configuration information tree. This service allows the system to boot up services from different locations, according to different organizational constraints, such as administrative domains, service providers, etc.

According to the invention, software controllers can be integrated into the configuration infrastructure according to a general design template for building software network controllers, and/or an extensible protocol for controller-to-controller negotiation and adaptation. A software distribution and downloading service allows distribution and downloading of controller code for remote installation of controllers and system components.

A monitoring system according to the invention comprises a set of management agents, preferably one per local computing device, and a Domain Manager, preferably one Domain Manager per network domain. The monitoring system allows visualization of the bootup or attachment process, such as determining which controllers are running in the network, their location and their state. It may also perform fundamental operations, such as launching, shutting down, or resetting a controller or a group of controllers. Preferably, it is designed to recover from the failure of any of its components, and ensure that the network configuration information will remain consistent despite failures; to reset the system or parts of the system to a consistent state; and to auto-configure itself

In certain aspects of the invention, if the Domain Manager fails, it will restore its knowledge about the state of the network by querying the directory tree after a failure to find the location of agents that were present in the network during the lifetime of the directory instance. The Domain Manager can then query the agents to obtain information about their local controllers to restore its knowledge of the state of the network. According to the invention, the Domain Manager does not need to store any persistent information itself, since it can obtain information about local controllers from the management agents.

Brief Description of the Drawing

The invention is described with reference to the several figures of the drawing, in which,

Figure 1 shows the dynamics of a nomadic device attachment;

Figure 2 shows the structure of the configuration tree;

Figure 3 shows the architecture of a controller;

Figure 4 shows the phases of a simple negotiation;

Figure 5 is a flowchart of the negotiation process;

Figure 6 shows an example of controller negotiation;

Figure 7 shows the relationship between sub-components of the software distribution service;

Figure 8 is a flowchart of the code submission process; and

Figure 9 is a flowchart of the code downloading process.

Detailed Description

Architecture overview

A multimedia network is a set of computing devices (computers, switches, routers, etc.) and associated software that provides quality of service (QOS) guarantees. A computing device is a piece of hardware that has network communication and/or processing capabilities. Attaching a new device to a network means granting the device access to the services already available in that network. The newly attached device may also advertise and make available its own set of services to the other network machines.

A service instance is a graph that has associated with it a set of network resources (buffer space, bandwidth and addressing), a media transport protocol, and a service management system. The service creation process consists of interconnecting together, or binding, a set of network resources in a seamless manner. This operation is performed by software entities called controllers.

The communication process in a multimedia network can be viewed as a result of coordinated manipulations of network resources (buffer space, bandwidth and addressing) by software entities called controllers. Controllers administer the network resources, or implement a local or distributed service on top of these resources. Controllers can operate either in cooperation or in competition with one another. A service in the multimedia network may be realized by the actions of one or more cooperating controllers. Multiple simultaneously executing instances of a service may result in controllers competing for the same resources.

In order to create or run a specific service, controllers need to interact between themselves. Interacting controllers form communities. When a computing device is attached or booted into a domain, it must be able to communicate and establish an inter-working relationship with the devices already running in that domain. According to the invention, a number of elements are used to create a configuration infrastructure supporting this process. These may include a configuration tree for organizing and sharing configuration, management and control information related to multimedia networks in a uniform and consistent manner; a domain directory service for publishing and retrieving the information; a set of local configuration trees (one per device) that allow local hosts to cache the device configuration settings and

remember the customized configuration attributes; a general design template for building software controllers so that they can be integrated with our configuration infrastructure; a general protocol for enabling controller-to-controller negotiation and delegation; a software distribution and downloading service for remote installation of controllers and system components; a watchdog controller that continually monitors that location and state of attachment of the client; and a control and monitoring system for the bootup and attachment processes.

The relations between these elements are illustrated in **Figure 1.** As shown, a service provider **10** on a multimedia network publishes information about a service (*e.g.*, availability, system requirements, pricing and subscription or purchasing information) to a directory service **12** via a schema. The schema serves to enforce uniform naming and logical data representation conventions across the local network **14.** A client **16** uses the directory **12** service to browse and select services of interest. Once a choice has been made, the client **16** uses the software distribution service to download and install the components that comprise the service from the service provider **10.** These components are preferably controllers developed according to a software template. The template specifies a set of standardized interfaces that each controller must implement so that it can work with other controllers and, if desired, be custom-configured by the client **16.**

When a user roams to a new foreign network **18** and attaches to it, a watchdog controller detects a change in the attachment location and initiates a re-configuration process to notify all controllers of this change. At the same time, the user may browse and select to install new or replacement services in the foreign network, using the foreign service provider **20** and foreign directory **22.** Whenever such an installation is initiated, components of the newly installed service might need to negotiate with servers of the service provider **10** in the home network **14** to update or establish operating parameters. They may make this contact via a controller-to-controller negotiation protocol. The entire bootup or attachment process is controlled and monitored by a system, comprising a Domain Manager and a set of management agents (preferably, one management agent per computing device in the network).

The software infrastructure of the invention allows booting up automatically or integrating already running controllers into a multimedia network. This process is realized in three steps: coordination, negotiation and adaptation.

Coordination is required if the controller must interact with a different set of controllers in the network because its location has changed, or because it was started, but it has no information about the network. Negotiation is required if the options and parameters supported by controllers in the foreign network are not compatible with those expected by the local controller. Adaptation is required if the types or values of the configuration parameters used in the network are different from those used by the controller. At times, it may be impossible for the controller alone to accommodate these changes and the presence of other intermediate controllers may be required to bridge differences. In such instances, the infrastructure preferably allows any required controllers to be downloaded onto the local device.

In operation, the controller first learns about the settings of the new network, and compares them to its own settings. If the settings are compatible, the controller is ready to run. Otherwise, it will negotiate, asking the network for additional information such as a range or a set of possible values for a specific attribute, and choosing, if multiple choices are present, the value that better suits its configuration. Finally, during the adaptation phase, the controller will make effective the changes that appeared to be necessary at the end of the negotiation process.

A computing device that is being started or is being attached to a network might advertise a new type of service, to be used by a set of network clients. This is referred to as service publishing. A centralized service is advertised to the interested clients, which may then invoke it remotely. For a local distributed service, the code is made available, so that every interested party can download the code and run the service locally. When a service is published, it is necessary to specify the type of the service, specify the service configuration parameters, and provide a capability for downloading the service code or remotely invoking the service.

Programmable networks allow third-party software developers to write network services on top of a set of network resource abstractions. Several implementations of the same service, offered by several providers, may run at the same time on top of the same network resources. Services may cross administrative

boundaries, overlap over several security, accounting and billing domains, or be offered over several Ethernet local area networks connected by bridges, gateways or routers. The network area covered by the service may not overlap in its entirety over a set of previously defined domains. Different services may come with specific security, accounting, or billing policies.

In order to efficiently implement these goals, service management and service configuration management according to the invention have a modular structure. There is a need to be able to provide separate configuration information repositories for services provided by different vendors, as well as the ability to bootup the same type of service from different locations. In order to solve these problems, our architecture provides the capability to specify where configuration information for each service may be obtained. A service can be associated with one or several configuration repositories. A repository can be defined per service. Alternatively, the controllers implementing the service can boot from local configuration units.

The controller attachment infrastructure comprises a monitoring system. A management agent monitors the bootup or attachment process locally at each network location. Management agents send requests and updates to the configuration servers from which their local controllers are booting. One management agent can contact one or several configuration servers in order to obtain the configuration information necessary to boot up its local controllers. Management agents could also perform additional operations such as launching, resetting, or shutting down controllers, but these operations could also be performed by other applications. A Domain Manager receives events from the management agents. These events allow the configuration process to be monitored, including monitoring the state of each controller launched in the network and detecting any alarms, errors, or exceptions that occur during the bootup, shutdown or execution of the controller. The Domain Manager may also initiate commands to shutdown, restart, or reset a controller or a group of controllers.

Architectural Components

The preferred embodiment of the system of the invention described herein comprises at least one service provider, a central directory server that stores default configuration information in a tree structure, at least one local management agent and

one local configuration database per computing device attached to the network, and a domain manager. Such a system is called a domain. Service providers may implement a security system whereby code is encrypted during transmission across public networks. This system may be facilitated by a separate software distribution service that maintains a certification authority. For preferred embodiments of the invention, there is further defined a design template for controllers so that third parties may create new controllers that will perform predictably for different clients on the network. For the preferred embodiments of the invention, all the instances of a distributed service booting inside a domain will have the same configuration parameters. For example, all the routing controllers will have the same timeout values.

The Network Configuration Tree Structure

The configuration information is stored in a hierarchical tree, as shown in **Figure 2.** The structure of the configuration tree reflects the hierarchical naming scheme adopted to identify controllers. Controllers are classified according to their class and associated with a class identifier. Moreover, individual instances of a controller are identified through a unique object identifier. Uniqueness is achieved by constructing the identifier from a combination of device address, process identifier and instance identifier.

Configuration parameters are either common to all the instances of a specific object type, or are specific to each instance of an object class. The configuration infrastructure is built according to this classification. The centralized domain directory tree **30, 32** contains the common attribute values **34** and the default instance attribute values **36** for each controller **38**, or group of controllers. The common attribute values are common for all the instantiations of a specific controller type. The default instance attribute values can be changed during each controller instantiation. The local configuration tree provides information about the instance attributes that have been customized by controller instantiations during their execution. It is also within the scope of the invention to store information about customized attribute values in the centralized domain directory tree, rather than at the

local hosts. In such embodiments, the information is preferably stored as a subtree whose first children are subtrees specific to each host.

Every time a new computing device (identified by its address or name) boots using information stored in the domain directory, it will also register with the domain directory server, which stores a subtree **40** of registered hosts **46**. If the values of any configuration parameters are changed during bootup, these modified values **42** will be stored in the local configuration tree **43**. As in the centralized domain directory tree, the customized attributes are stored under nodes for each controller **44**. However, only attributes that have been changed from their default values need be stored in the local configuration tree. The entire configuration information is preferably also be cached in the local configuration tree.

In this way, a default template configuration, common to all the applications of a specific type, is provided. One also has the capability to overwrite the values of one or several specific attributes for a controller instantiation. The approach is somewhat similar to an object-oriented programming approach: a base class defines a set of methods and all the inheriting classes can either overwrite the implementation of the methods or use the implementation provided in the base class.

The architecture allows new services deployed in the network to advertise their configuration information. When a new controller type is run for the first time in a network, it can advertise its configuration by setting up the default attributes in the default subtree of the database. After that, any other instance of this controller type can boot up using the default configuration information. Once the controllers are running, they can customize their configuration, by overwriting some default configuration attribute values, and storing the customized configuration locally.

The configuration architecture provides the capability of defining controller groups, and associating configuration parameters with the newly formed groups. During its execution, a service can interact with other services, or with the controllers that administer certain network resources. Because of these interactions, it can be desirable to form a group of controllers by running several controllers together in a single process, or in another container type.

The value of any configuration attribute is entered only once in the domain directory. The value of a specific configuration attribute is specified only when it is

referred to for the first time. Any further references will be pointers to the first reference.

The Domain Directory Service

The Domain Directory Service is the front-end to the domain configuration tree. It provides the following functions:

It provides upon request configuration information to the devices booting up in the domain.

It registers all the devices that have boot up in the domain in a repository.

It uses the registration repository information to notify all the devices attached to the domain about a change. For example, if the location of a service changes, the service can notify the Domain Directory Service. Upon receiving the notification, the Domain Directory Service will update the domain configuration tree, and then it will forward the change to all the devices present in its registration repository. If a device is up, it will receive the notification. If it is down, the notification will be lost. This procedure ensures that all the devices already running in the domain will be aware of the change. Any devices that will boot up later will retrieve the change in the modified configuration parameters.

A Domain Directory Service interacts with a set of management agents (preferably one management agent per computer), a set of local configuration databases (preferably one local configuration database per computer), a Domain Manager (which may interact with multiple configuration units), and a set of controllers launched on the machines booting from the configuration unit. The controllers are not part of the configuration infrastructure, but are its clients.

Management Agents and Local Configuration Databases

The local configuration database is a tree with three branches. The first branch contains local configuration. The second branch contains the configuration attributes that have been customized during the last bootup. The third branch contains a cache of the configuration during the last bootup.

The local configuration database is a hierarchical database containing two kinds of information. First, it contains a list of configuration server locations and the

types of controllers for which they have information. It also contains a cached copy of the last configuration that was run on the local machine. The cached configuration will typically consist of the management agent configuration, the names of the controllers or groups of controllers that were run on the local machine and all the values of all the attributes the management agent and the controllers used to configure themselves.

There are three ways that a device may boot up in this configuration:

1. Bootup using the default parameters: the controllers launched on the device will be configured using only the default configuration attribute values stored in the domain directory.

2. Bootup using the last cached customized configuration: the device contacts the domain directory server, gets all the default configuration attributes, checks which of these attributes were overwritten when it booted up last time, and overwrites them again. In this way, the device will be started with the same customized configuration parameters. However, if any of the default parameters that are not overwritten locally have been changed since the last bootup, those changes will be effective during the current device bootup.

3. Local bootup: the device will not contact the directory service, but will simply boot up using the configuration it cached locally last time when it was connected to the domain.

Management agents can support the local system configuration process in a number of ways. They may provide the capability to specify from where applications should boot, and start the booting process, and may provide a local boot capability when configuration servers are unavailable. They may read a previously cached configuration from a local hierarchical database, ask the directory server for configuration information, launch the local applications, or allow the applications to be launched manually, or by other means, and/or provide a mechanism for monitoring the bootup and the shutdown. Finally, they may process events received from the local applications, for example to let the local user know about the status of a local application or an alarm, or send these events to the Domain Manager.

The Domain Manager

The Domain Manager can support the local system configuration process by receiving events from the management agents to monitor the bootup process; resetting the system or parts of it (by killing all the applications that were launched on that part of the network and were registered with the system), and/or sending information about the monitored network to several graphical user interfaces

Design Template for Building Controllers

Controllers are not part of the configuration infrastructure. However, they are interacting with it, and they receive specific requests from the management infrastructure, such as instantiating or configuring themselves. These requests are sent through a management interface that includes functions such as instantiation, destruction, configuration, negotiation and delegation. Each controller implements a management interface that will be invoked by the configuration infrastructure. The management interface specification is part of the design template **50** of a controller **52**.

A controller **52** is the smallest modular unit of software that can be independently downloaded, instantiated, configured, managed and/or executed. A controller is characterized by the primary service it provides. Users request this service through a set of one or more interfaces **54** that the controller implements. These interfaces are known as operational interfaces because they realize the operational nature of the controller. Each operational interface in turn may contain one or more functional methods. For example, a connection controller providing connection setup services may implement two interfaces, one for requesting connection establishment and another for setting parameters that determine how its connection establishment algorithm should operate. The first interface may in turn contain two methods, one specific to unicast point-to-point connection and one for multicast.

The software code implementing the controller is called the controller implementation. Because controller implementations deal only with the functionality of the operational interfaces, they need to be augmented with additional interfaces **56** to support controller-to-controller negotiation, configuration, initialization and

downloading. This can be achieved by specifying a separate controller template class that includes these additional interfaces from which the controller implementation must inherit. Some methods in these interfaces are abstract; hence, invocations on these will be directed to the corresponding concrete methods in the controller implementation. Finally, controllers can also emit events or notifications either as part of their operational behavior or in response to exceptional conditions. These emissions can be channeled to the interested receivers for their appropriate response.

Controllers are instantiated by controller factories **58**. These are specialized controllers whose primary role is to create other controllers. Controller factories implement a single pre-defined factory interface **59** as their sole operational interface. With the exception of this feature, controller factories do not differ from other controllers. By adopting this design, controller factories themselves can be dynamically instantiated by other factories or downloaded on the fly to create a needed controller. **Figure 3** shows the relation between a controller implementation **52**, a controller template **50** and a controller factory **58**.

Initialization, configuration and management interface

The initialization, configuration and management interface **60** consists of methods that allow a controller factory to initialize and configure a controller. Instantiating a new controller involves locating the code base of the controller, downloading the dynamic link library (DLL) that implements the controller and finally obtaining and calling the entry point of the controller's constructor. All constructors must take a single argument composed of a sequence of name-value attribute pairs. The name-value attribute pair structure allows a factory to pass through the constructor a variable length combination of any elementary data types in an unambiguous manner.

The configuration methods of the template interface allow re-configuration of the controller after instantiation. Again, the parameters for re-configuration are expressed as a sequence of name-value attribute pairs. Finally, the management methods of the template interface allow an external entity to monitor the state of the controller. Monitoring can be achieved in 2 ways. In the polling method, the monitoring entity periodically polls the management interface of the controller for the

appropriate state information. In the interrupt-driven method, the monitoring entity sets the event emission mechanism of the controller to emit events to its receiving interface and subsequently waits for event triggers.

Negotiation and delegation interface

The negotiation and delegation interface **62** allows a controller to initiate and participate in a negotiation procedure with another controller or to request a controller to delegate a call to another controller. In both cases, a method in each of the interfaces is associated with a corresponding method in one of the controller's operational interfaces. For the former, the method is called a proposing method, for the latter, a delegating method. Details associated with the use of these methods are described hereinafter.

Programmability interface

Controllers that are large or implement complex functionality may be broken into further sub-components for better modularity and reusability. The programmability interface **64** may allow these sub-components to be dynamically changed or upgraded during the execution lifetime of the parent controller. The interface may include methods for identifying sub-components in the controller, loading a sub-component, locking the controller's execution state so that a sub-component can be safely replaced, and unlocking the controller's execution state so that program execution can continue.

Reflection interface

The reflection interface **66** allows a caller to enumerate the list of all interfaces implemented by a controller and to discover the structure of each interface. "Structure" may include:

- For each interface, a list of method names
- For each method name, a list of argument names
- For each elementary-typed argument, its type
- For each complex-typed argument, a recursive list of its children

In addition, the reflection interface also allows a controller to declare a static list of other types of controllers it needs to call in the process of its execution as well

as a mapping table for associating methods in the operational interfaces with the appropriate proposing or delegating methods.

Controller Negotiation and Delegation Protocol

Since controllers are expected to run in diverse environments including those not envisioned when they were originally designed, traditional static procedure calling mechanisms might be insufficient. In such mechanisms, there is an implicit assumption that the caller has full knowledge of the parameter set of the calling interface including the semantic meaning of each argument. Moreover, it assumes that a called controller will always either accept all the arguments and execute the call or fail the call because some of the arguments are not acceptable. In essence, the approach is in line with a master-slave model of interaction where the caller is assumed to be more intelligent and delegates tasks to the callee. The model can also be extended to allow a caller to delegate tasks to a third party through the callee. In the alternative approach, a peer-to-peer model is assumed where pairs of controllers interact with each other at the same level.

To realize both approaches we define two types of methods known as proposing and delegating methods in the negotiation and delegation interface.

Proposing methods

A caller of an operational interface method who is unsure of the parameters to use for the call will first invoke the associated proposing method on the negotiation interface. The proposing method allows a caller to initially propose a desired parameter set. The response of the method will be either acceptance or a counter-proposal by the called controller. In either case, the corresponding method associated by the operational interface with the proposing method will be indicated in the response. If the caller accepts the counter proposal, it simply calls the indicated method and the negotiation phase ends with a success (see **Figure 4**). Otherwise, there are two possible outcomes. If the caller is prepared to renegotiate based on the counter proposal received from the callee, it simply re-calls the proposing method with the new parameter set. If the caller is not prepared to renegotiate, it simply ignores the counter proposal and the negotiation phase ends with a failure. In the former case, the cycle of proposal and counter-proposal can be repeated as many

times, as the caller desires. In all cases, it is the caller that makes the final decision to accept or reject the negotiation process. The outline of this process is depicted in **Figure 5**.

Delegating methods

Delegating methods extend the standard procedure-calling model by allowing the initiator to delegate the call to a third party via the callee. In terms of syntax, the delegating method simply adds an additional parameter (the third party's identity) to the original method. The called controller examines this parameter and if acceptable forwards the call by invoking the appropriate method on responsible party's operational interface. As in the model for negotiation, the delegation process can be daisy-chained to arbitrary lengths by having subsequent controllers re-delegate the calls in turn. One possible use of this scheme is to implement multi-layered access control where controllers in each layer can only invoke methods of controllers in the subsequent layer. In such a case, security credentials are checked at every invocation request point.

In both delegating and proposing methods, a calling controller can determine the appropriate proposing or delegating methods by querying the reflection interface of the target controller.

Example of service requiring controller negotiation

In this section, we present a concrete example of a service that could require controller negotiation. Consider a connection control service with call forwarding as illustrated in **Figure 6**. In this figure, User-A is attempting to connect to a roaming user, User-C but is unaware of User-C's current location. Its local controller, Controller-A contacts Controller-B, the controller at User-C's last known location to request for a connection setup . However, because Controller-A not aware of the capabilities of Controller-B, it uses a proposing method CanConnect() to probe Controller-B. For convenience of notation, we assume controllers A, B and C reside on hosts A, B and C respectively.

In scenario 1, Controller-B replies with a positive acknowledgement that it can complete the call and tells Controller-A to use Controller-B's Connect() method. Controller-A acts accordingly and Controller-B sets up a segment of the call from

host B to User-C. Controller-A then sets up the final segment of the call from host A to host B and splices the two connections into one.

In scenario 2, Controller-B replies with a negative acknowledgement and gives Controller-A User-C's exact location (at host C). Based on this feedback, Controller-A decides not to involve Controller-B further and instead sets up a single 6-hop connection between User-A and User-C directly.

In the final scenario, Controller-B replies with a request that Controller-A delegates the connection control procedure to it by calling its ConnectX() method. Controller-A accepts B's counter-offer and calls Controller-B's ConnectX() method with the parameters host A and User-C. Controller-B then sets up the 6-hop connection from User-A to User-C.

Note that only in scenario-2 does Controller-A need to know the location of User-C. In the two remaining scenarios Controller-B uses full or partial delegation to hide the address of User-C from Controller-A.

The software distribution and downloading service

A software distribution service may be provided to ensure the reliable and secure transmission of controller code across open networks. The system preferably provides means to verify that all downloaded code is tamper-free and originates from the claimed publisher. Furthermore, the system should be able to irrefutably establish the identity of a user downloading code or of the code publisher to prevent unauthorized access.

The architecture of the software distribution system comprises a code publisher, a code repository (that manages secure storage of code), a certification authority (that issues and verifies digital credentials), a security manager (that registers code users and publishers), and one or more code users.

Figure 7 illustrates the relationship between the above components. A code publisher and code user must first register with the security manager for permission to publish or download code. Next, the publisher must obtain a digital credential from a certificate authority before it is allowed to publish content to a code repository. Code repositories on the other hand, use confirmations from certificate authorities and

security managers to validate requests for code publication or downloads from publishers and users.

The downloaded software may need to be checked for dependencies: often an executable cannot run without an underlying set of dynamic libraries. Once a client has downloaded an executable or a library from a software repository, it may to check if it has any dependencies, and if these dependencies are present on the local system. If the dependencies cannot be found, the client will have to request them from the software repository in the same way as it has requested the initial downloaded software.

To prevent tampering and interception of content, all communications between the 5 entities are protected via a public key encryption system. In this system, every communicating entity is endowed with a public/private key pair. Content encrypted with a public key can only be decrypted using the corresponding private key and vice versa. Public keys are openly published while private keys are kept strictly secret. For two entities to communicate securely they must first obtain each other's public key. Thereafter, any message being sent from one to the other is appended with the sender's public key before being encrypted with the receiver's public key. In this way, only the receiver can decrypt the message since only the receiver has its secret key. Moreover, since the sender has also included its public key in the message, the receiver can reply securely by encrypting the reply message using this key.

To irrefutably prove one's identity in the system, a digital credential such as a digital signature is required. Such signatures are generated by certification authorities using a combination of their signatures and the public key of the requesting entity. Certification authorities in turn obtain their signatures from other high-level certification authorities, thus creating a chain of verifiable identities. At the root of the chain, is the top-level certification authority whose signature is well published and hence requires no verification.

Figure 8 illustrates the code submission process. First, a publisher signs the code to be submitted with its signature **S1** before encrypting it with the repository's public key **P2**. The encrypted package is then sent to the target code repository, which uses its private key **K2** to decrypt its contents. The repository verifies the authenticity of **S1** and then queries the responsible security manager to verify that the

publisher has the rights to publish to this repository. If so, the repository stores the code along with the publisher's public key **P1** for later retrieval.

Figure 9 illustrates the code retrieval process. A user requests for a download by submitting **P1**, the public key of the publisher and **P4**, its own public key encrypted with the repository's public key **P2**. The repository decrypts the request using its own private key **K2** and checks its store for **P1**'s code. If the code is found, it verifies with a security manager (using **P4** and **P1**) that the user indeed has the right to download the code. If this is confirmed, the repository first encrypts the code using **P4** and then sends it to the user who will in turn decrypt it using its private key, verify its authenticity using **S1** (the signature of the publisher) before executing it. Since the code was signed with **S1**, the signature of the publisher, it cannot be subsequently tampered with without compromising the signature.

Architectural Dynamics

A system according to the invention preferably can retrieve, update, add, or remove configuration information in a directory, can create, monitor, or destroy a controller or controller group, and can change the configuration of an application (domain manager, management agent, controller, controller group, or the configuration system itself).

Communication Between Components

In one embodiment of the invention, the bootup process is monitored using an event system. The Management Agents receive state information from the local controllers. The agents process and send these messages to the Domain Manager. The events can be sent, for example, by TCP connections between agents and the manager, UDP datagrams sent between agents and the manager, via an RPC mechanism, such as CORBA, Java RMI or DCOM, or via an event channel.

In one embodiment, the configuration server has a CORBA interface. The configuration requests are sent to this interface by the management agents, or by the controllers. The configuration information is stored in XML documents.

The management entities may use CORBA to access the managed controllers. Every managed controller has a management interface, on which it receives requests from the components of the management architecture.

Controllers interact with each other via an exchange of 'signaling' messages that may be carried via inter-process communication channels (when both controllers reside on the same computing device) or via network communication channels (when both controllers reside on different parts of the network).

Bootup

At bootup time, the first system component to come up is generally the configuration directory server. The remaining objects need to know the location of the configuration server. The Domain Manager comes up after the configuration server. After the Domain Manager is up, it builds an internal image reflecting the current network state by storing the events sent by the system components.

The management agents are preferably launched before the managed controllers, so that they can receive, process and transmit further the events from the local controllers. The management agents may also use the directory to configure themselves.

The agents can launch the controllers, or the controllers can be launched by some other means (for example, via a script, or by hand).

Runtime Operations

The configuration server, the Domain Manager, and the management agents allow launching and configuring the managed controllers. It is also possible to monitor the bootup process. The monitoring process consists of visualizing the state of each controller, observing parameter values assigned to each controller, and reading any error messages that might be raised during the bootup. The management system also allows shutting down or resetting a controller or a group of controllers.

A controller can enter at least four states during its execution:

- Booting up – while it is booting up (reading its configuration and instantiating its internal variables, resolving references). The controller is not operational in this stage.
- Up – when it has booted up successfully and entered a state where it can receive requests from outside and process those requests.
- Down – when it has encountered an error, either during bootup or during normal operation.

- Shutting down – when it is shutting down (deleting its internal structures, releasing any remote references) due to a normal exit request.

An additional state, undetermined, may be associated with a controller by an external observer who cannot determine exactly its state, given the information he has about the controller on his local display or in his database.

An application can boot up locally, or over the network. The information necessary for bootup can be provided from a network directory, from a script, or from a database located on the local machine.

When a machine connects to the network for the first time, the template set of configuration parameters stored in the default branch of the network directory is used to initialize the local controllers. Each time a new machine is connected to the network, saves its configuration, as well as any configuration attributes that have other values than the default, in the local configuration directory. The content of the directory trees is preferably represented as an XML document. Any configuration constraints can be enforced using the XML Document Type Definition (DTD) framework.

Shutdown

The application shutdown process can be launched from a centralized location, from a local agent, or by simply killing the application. The event passing mechanism will allow managers at different levels (network, local machine) to learn the current status of the application (shutting down). Once the shutdown process is complete, the application and its resources are freed and are no longer tracked by the management system. The network may be reset to a consistent initial state by shutting down all its components, and then restarting each component.

Failure Detection and Recovery from Failures

Domain components preferably can interact with each other and with the managed applications in order to detect faults in the system, and to recover from those faults.

Domain Directory or Domain Directory Service Failures

The configuration server detects unit database failures when it receives an error code after sending a request to the database. Configuration Server failures are detected when a client does not receive a reply to a configuration request it sent to a configuration server. The configuration server preferably backs up the contents of the domain configuration database every time a new device registers with the configuration server.

Management Agent Failures

The Domain Manager monitors the Managements Agents inside the domain. The Domain Manager pings periodically every Management Agent. If the Domain Manager does not receive a reply from a given Management Agent after sending a given number ping messages, it sets an alarm state on that Management Agent.

The Management Agents can also monitor each other in order to discover any potential failures. Details of network monitoring are extensively discussed in U.S. Provisional Application No. 60/157,965, entitled "Dynamic Programmable Routing Architecture with QOS Support," which is incorporated herein by reference.

Domain Manager Failures

As described in more detail in U.S. Provisional Application No. 60/157,965, the Domain Manager does not store itself any persistent information. Instead, each management agent maintains information about local controllers. When the Domain Manager comes up after a failure, it queries the directory tree to find out the location of the agents that have registered with the network directory. The Domain Manager then queries each agent on the list about the state of its local controllers, and using this information, it restores its knowledge about the state of the network.

Managed Applications Failures

The management agents may provide the capability of monitoring the local applications, processes or controllers by receiving heartbeat messages from them or by polling them.

Presenting a Consistent View of the System to an External Observer

The state of the network can be monitored from a management graphical user interface. The information on this interface should reflect accurately the state of the network it is representing. If not enough information is available to determine accurately the current state of a network component or domain, it should preferably be represented in an undetermined state, that will let an external observer know that something might be wrong in that area.

Other embodiments of the invention will be apparent to those skilled in the art from a consideration of the specification or practice of the invention disclosed herein. It is intended that the specification and examples be considered as exemplary only, with the true scope and spirit of the invention being indicated by the following claims.

What is claimed is:

1. A configuration infrastructure supporting attachment of computing devices to a multimedia network domain, the computing devices running controllers, the configuration infrastructure comprising:
    a domain directory service, comprising:
        a domain configuration directory that stores configuration information for controllers in a tree structure, the tree structure comprising default instance attribute values for controllers; and
        a domain directory server that serves as a front end for the domain configuration directory; and
    local configuration directories that include information about customized instance attributes that differ from the default instance attribute values.

2. The configuration infrastructure of claim 1, wherein the local configuration directories are stored as subtrees of the domain configuration directory.

3. The configuration infrastructure of claim 1, wherein the local configuration directories are stored on the computing devices.

4. The configuration infrastructure of claim 1, wherein the domain directory comprises default instance attribute values for each controller as a child of the root.

5. The configuration infrastructure of claim 1, wherein each computing device contains a local configuration tree, said local trees comprising customized instance attributes for local controllers.

6. The configuration infrastructure of claim 5, wherein the configuration information is generated by taking the default configuration for the controller from the domain directory and overwriting any customized instance attributes corresponding to the controller from the local configuration tree.

7. The configuration infrastructure of claim 1, wherein the computing devices query the domain directory service to obtain configuration information for a local controller at instantiation.

8. The configuration infrastructure of claim 1, wherein each computing device stores a local configuration database that comprises configuration unit locations for controllers run on the devices.

9. The configuration infrastructure of claim 8, wherein each computing device stores a cached copy of its last configuration.

10. The configuration infrastructure of claim 8, wherein each computing device stores the default attributes that have been assigned customized values.

11. The configuration infrastructure of claim 8, wherein each computing device stores its local configuration.

12. The configuration infrastructure of claim 1, wherein each computing device runs a local management agent that communicates with local controllers and with the domain directory service.

13. The configuration infrastructure of claim 12, further comprising a domain manager that monitors the state of all local management agents in the network.

14. The configuration infrastructure of claim 13, wherein the domain manager, upon restarting after a failure, queries the domain directory service to locate the management agents in the domain, and then queries the management agents to determine the state of the controllers at each computing device.

15. The configuration infrastructure of claim 13, wherein the domain manager periodically pings the local management agents, and wherein the domain manager sets an alarm state at a local management agent when the local management agent does not respond to a predetermined number of pings.

16. The configuration infrastructure of claim 1, wherein
the domain further comprises at least one software distribution service;

computing devices find the location and parameters of the software distribution service by querying the domain directory service; and

computing devices obtain software directly from the software distribution service.

17. The configuration infrastructure of claim 16, wherein the computing devices obtain software by remotely invoking it from the software distribution service.

18. The configuration infrastructure of claim 16, wherein the computing devices obtain software by downloading it from the software distribution service to run locally.

19. The configuration infrastructure of claim 18, wherein the downloaded software is encrypted during downloading and subsequently locally decrypted by the computing device.

20. A multimedia network domain, comprising:

a service provider;

a directory; and

a service user, wherein

the service provider publishes to the directory information about an available service;

the service user accesses the directory to acquire the published information; and

the service user contacts the service provider to install the available service.

21. The multimedia network domain of claim 20, wherein the network comprises multiple service providers, each publishing information to the directory, and wherein the service user can select which service providers to contact by accessing the directory.

22. The multimedia network domain of claim 20, wherein the service user may roam to a foreign network, and may renegotiate with the service provider to adjust the settings of the provided service for the foreign network.

23. The multimedia network domain of claim 22, wherein the service user may also use services provided by service providers on the foreign network.

24. The multimedia network domain of claim 20, wherein the directory stores configuration information for the service.

25. The multimedia network domain of claim 24, wherein the configuration information is stored in a tree structure, the tree structure comprising a subtree containing default instance attribute values for controllers.

26. The multimedia network domain of claim 25, wherein information about customized instance attributes that differ from the default instant attribute values is stored in a tree structure local to the service user.

27. A controller running on a computing device attached to a network, the controller comprising:
a reflection interface that returns a caller a list of method names, argument names, argument types, or argument children for the controller; and
a negotiation interface, comprising
a proposing method wherein the controller responds to a proposed set of calling parameters by either accepting the parameter set or returning a counterproposal set of parameters.

28. The controller of claim 27, wherein the controller further comprises a delegation interface whereby a caller can direct the controller to invoke a method at a third party.

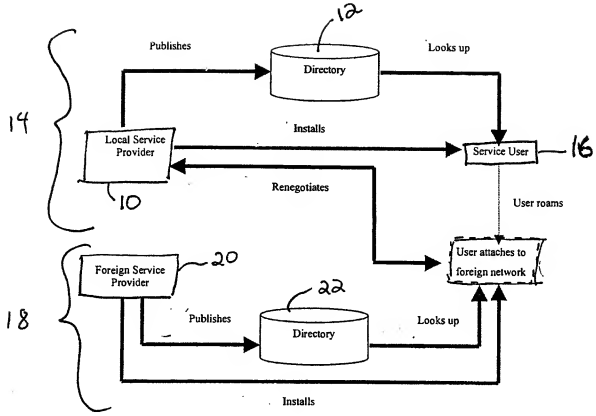


Figure 1: The dynamics of a nomadic device attachment

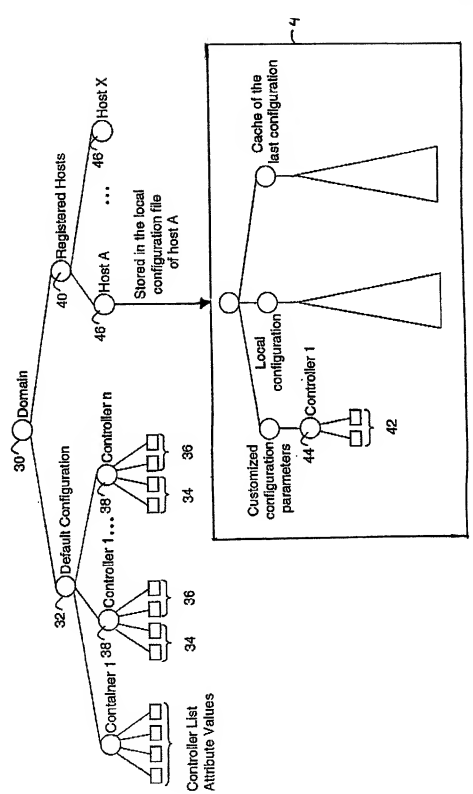

Domain
30
Default Configuration
32
Container 1
38'
Controller 1
38
Controller n
38'
Controller List
Attribute Values
34
36
34
36
Registered Hosts
40
Host A
46
...
Host X
46'
Stored in the local configuration file of host A
4
Cache of the last configuration
Local configuration
Customized configuration parameters
Controller 1
44
42

Figure 2

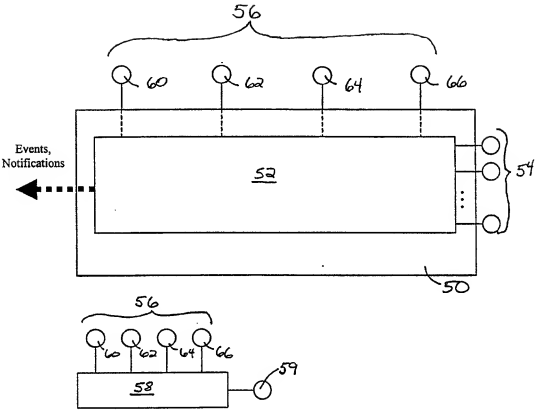


Figure 3: Architecture of a controller

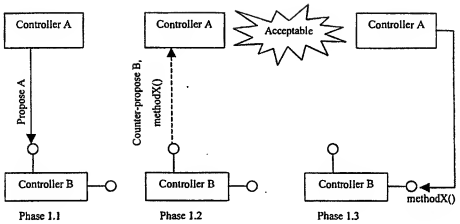


Figure 4: Phases of a simple negotiation

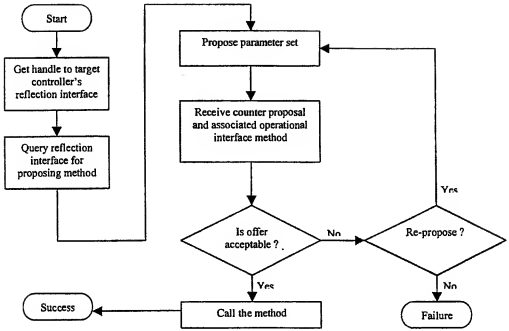
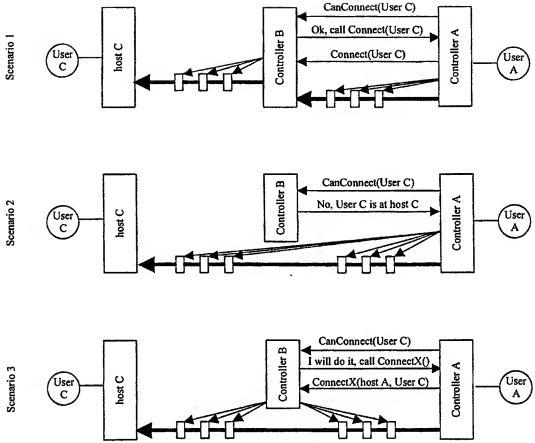


Figure 5: Flowchart of the negotiation process



Figute 6: Example of controller negotiation

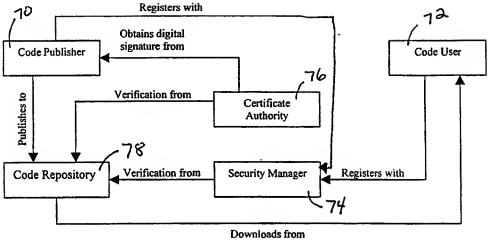


Figure 7: Relationship between sub-components of the software distribution service

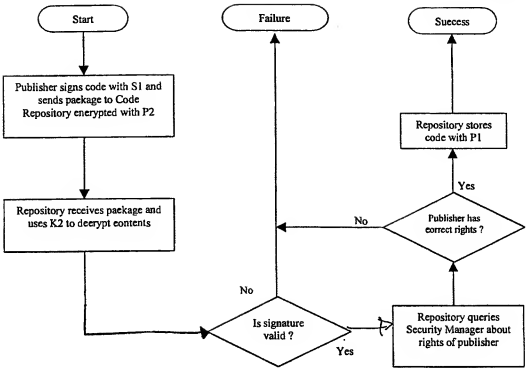


Figure 8: Flowchart of code submission process

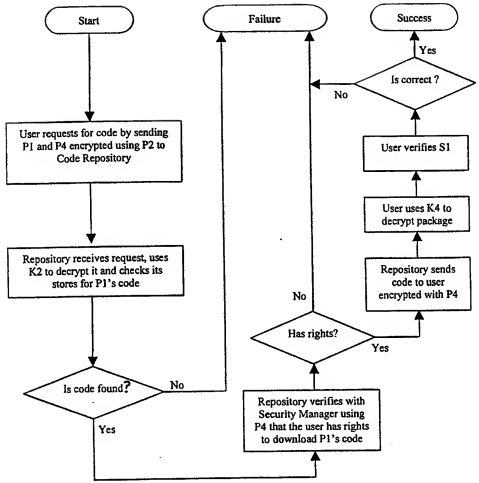


Figure 9: Flowchart of the code downloading process